

PROGRAMMING EXPERT

FAQ

Q How do I get started in games programming?

A The PC is a great platform to learn games programming on – it's cheap and you don't need a licence to sell your games. Games are some of the most complex and demanding programs to write, though. You need to know about 3D graphics, animation, sound, music and even physics – plus how to program them all.

The performance demands of games also mean you must master an efficient language, usually C or C++. Generally this is too much to handle in one step. Try mastering an easier language such as C# first. Then learn how to use DirectX, the Windows multimedia system. Once you have knowledge of these, you can begin to create working graphical games as you study all the trickier stuff.

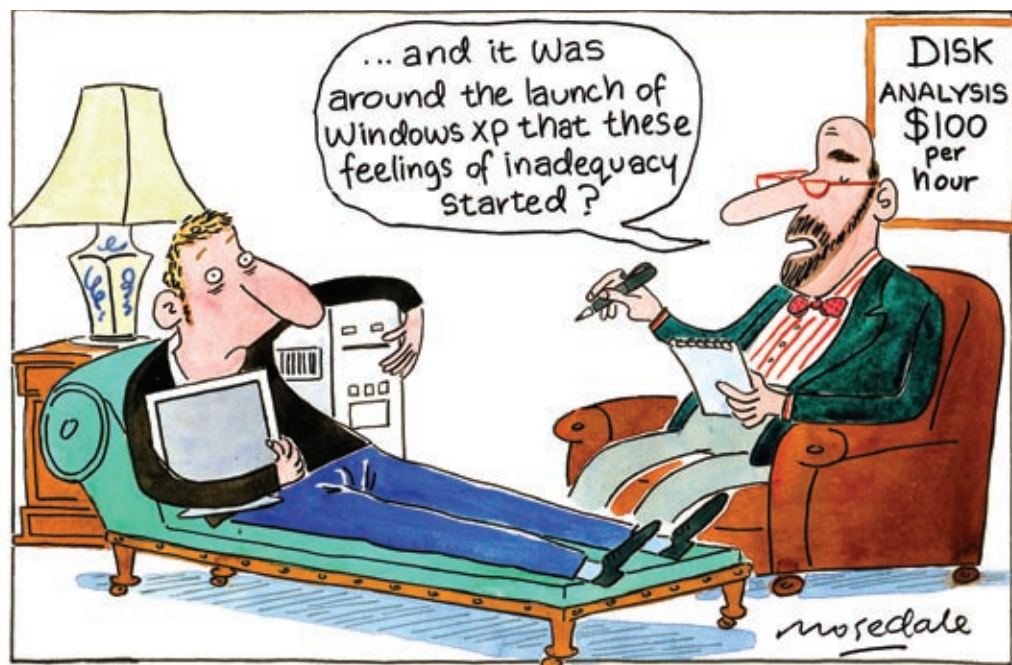
Mike James

IT author, lecturer and programmer

mike@computershopper.co.uk

Disk analyser

With the right programming you can take a cluttered hard disk in hand. Mike James explains how to write a disk analysis tool



Today's hard disks have so much space that managing one is becoming increasingly difficult. A well-used modern PC will have more than 50,000 files in over 5,000 folders, occupying 80GB or more. Unless you are super-organised, you'll need help finding documents on a PC this loaded – which is why Google, Yahoo! and MSN all offer desktop search facilities. An even thornier problem awaits when your big hard disk finally fills up, though. How do you work out where all your gigabytes have gone and find those files whose removal will free up the most space?

This month's project is a disk analyser that will show you where those big directories are lurking. You can develop and extend it to provide a wide range of information on the contents of a storage device. In our project we'll learn how to use the TreeView control, solve problems using recursion, work with files, directories and drives, use the Progress Bar and add images to a TreeView. You can find the complete program on the cover disc.

ANALYSE THIS

The first thing a disk analysis program needs is a user interface. This lets you pick a drive to analyse and then presents the results. The ideal control to use for picking the drive would be the DriveCombo box in Visual Basic 6. However, we want to use the C# .NET language for our project, because you can download it for free from <http://msdn.microsoft.com/vstudio/express/visualcsharp>, and this doesn't have an equivalent to the DriveCombo box. As luck would have it, though, we developed a custom C# control that does exactly the same job in last month's *Computer Shopper*. The code for the control is included both on last month's and this month's cover disc ready for use.

Start a new Windows Application in C# Express Edition and call it DiskAlyse. To add our custom control, you must do two things. First locate the folder containing DirCombo.cs on your cover disc and copy it to the project folder. Then choose Add Existing Item from the Project menu, and select the file DirCombo.cs.

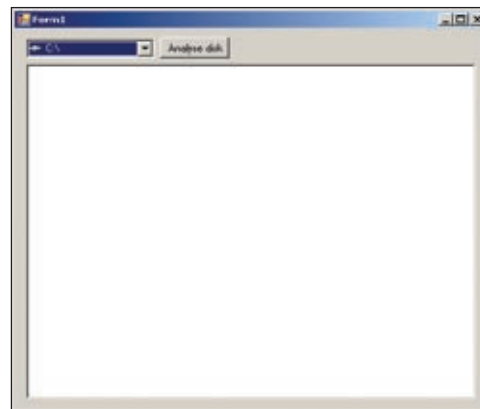
If you now build the project you will see that a new control has appeared in the toolbox. Simply place an instance of this control on the form, and you'll have provided a way for the user to select a drive to analyse.

To complete the user interface, simply add a single button and a TreeView control to the form. The TreeView control is useful for all sorts of tasks and is bound to become one of your favourite tools in time. You can use it to display a hierarchy of items, or to navigate them. This makes it perfect for representing the hierarchy of directories on a hard disk.

ROOT AND BRANCH

It always helps if you write a simple program before moving on to anything more complicated. Our first program will simply analyse how much room each directory on a disk uses and represent the results visually.

Our program needs to start at the root directory, find all the files that are stored there and work out the total space they occupy. But the root directory has subdirectories, which contain further files and further directories. So to get the full size of the root



▲ The user interface for DiskAlyse



directory we need to include the size of the files in the subdirectories, the subdirectories' subdirectories and so on. Sizing each subdirectory involves the same code as sizing the root directory. The part of the program that does the sizing can simply call itself to size any subdirectories – a programming technique known as recursion. If you think this sounds complicated, don't panic; recursive methods are much simpler in practice than in theory.

To start our program, double-click the button to open its click event handler and begin with the lines:

```
private void button1_Click(
    object sender, EventArgs e)
{
    string RootDir = dirCombo1.Text;
```

This stores the root directory selected with the dirCombo1 control in the RootDir variable, ready for us to work on.

The next step is to convert this string into the corresponding DirectoryInfo object, which in turn gives us access to the directory's properties, as well as the files and other directories it contains. To create this DirectoryInfo object we simply use:

```
new DirectoryInfo(RootDir);
```

As we explore the directory hierarchy, we need to keep a note of the directories we have visited. The obvious way to do this is with the TreeView control, but there are a few minor problems. So first we need to look at how the TreeView control works.

TREEVIEW

The TreeView control contains a collection of TreeNode objects, and you access this collection

using the control's Nodes property. Initially, a TreeView control doesn't have any nodes, and most of the work in using a TreeView control involves adding nodes.

Each TreeNode object has its own collection of TreeNode objects. By nesting nodes within nodes, you can create hierarchies that are as complicated as you like. In most cases, there will be a single top-level TreeNode stored in the TreeView control and various levels of child TreeNodes below this.

As the TreeNode object does everything concerned with creating the hierarchy, you may be wondering why you need to bother with the TreeView object. If you only want to create a hierarchy and are not interested in displaying it, you can use just TreeNode objects. But if you need to display the resulting structure and perhaps enable users to interact with it, you need to store the root TreeNode in a TreeView control.

The TreeView object displays all the TreeNodes it contains as a tree diagram and allows the user to move through the structure. The user can expand and contract branches of the tree by clicking on plus and minus signs. Each TreeNode's Text property is displayed on the corresponding node of the tree diagram, but it is also possible to display a picture, more of which later.

Being able to organise text and even pictures into a hierarchy isn't enough for many applications. Programmers often go to great lengths to make the hierarchy of objects they really want to use fit in to the text or picture categories. This is quite unnecessary. Every TreeNode object has a Tag property, which is of the type Object and so can be set to reference anything.

In our case, we need to keep track of a hierarchy of DirectoryInfo objects, and we can do this using the Tag property. The tagged objects

aren't shown when a collection of TreeNodes is displayed in a TreeView control.

RECURSION EXCURSION

With a basic understanding of the TreeView and TreeNode objects, we can write a recursive program that works its way down the tree from the root directory; if you want to think of it as working up the tree from its root, you can.

Writing a recursive program is much easier if you pretend you have already solved the hardest part and worry about its details later. For our problem, if we could wave a magic wand, we'd create a function that examined a given directory and reported the total amount of storage all its files and subdirectories use. Specifically, we'd want a function like this:

```
subTree(TreeNode)
```

This returns the total storage used by all the directories, starting at TreeNode and working down the hierarchy.

Assuming we had such a function, the Analyse button click event handler would begin as follows:

```
private void button1_Click(
    object sender, EventArgs e)
{
    string RootDir = dirCombo1.Text;
    TreeNode Root = new TreeNode();
    Root.Tag = new
        DirectoryInfo(RootDir);
```

This creates a new TreeNode object, Root, to store the details of the root directory, and uses its Tag property to store the DirectoryInfo object associated with the root directory.

We need to add the new TreeNode to the TreeView control as follows:



MICROSOFT PRESS

Visual Web Developer: Build a Web Site Now!

★★★★

£9

This is another in the Microsoft Press series of 'Build something or other now!' books. The best feature of these books is that they include appropriate Express versions of Microsoft development products – in this case, Web Developer 2005 Express Edition, together with SQL Server 2005 Express Edition. As both products are currently available for download, we have to ask what this book adds. Sadly, it adds little.

It's far too short to cover even a fraction of the ideas involved in web development. It is colourful and enthusiastic, but author Jim Buysen spends precious pages discussing things you should already know and may consider off-topic. The book is a cheerful companion to the CD, but if you want to learn anything useful about creating ASP .NET websites, you'll need another book.



SUMMARY

- ☒ Web Developer 2005 EE
- ☒ Book adds little to the disc

SUPPLIER www.amazon.co.uk
ISBN 0735622124
PAGES 192

PARAGLYPH PRESS

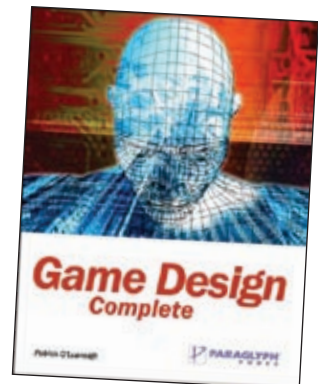
Game Design Complete

★

£18

To consider this book on games design complete, you'd need a particular understanding of the word 'design'. In my world, design means creating animations, organising resources and implementing 3D, but Patrick O'Lunaigh's book deals with topics that are more in the realm of Lawrence Llewellyn Bowen. This might be OK if you are a creative designer with a team of competent programmers at your disposal, but much of the text is just a commentary on common sense. It is also littered with poor-quality screenshots, which is particularly disappointing in a design-focused title.

A code-free book on game design is an interesting idea, but this seems to be out of touch with the technology, and is at best only slightly familiar with the business side of game creation.



SUMMARY

- ☒ Not too technical
- ☒ No practical use

SUPPLIER www.amazon.co.uk
ISBN 1933097000
PAGES 430

```
treeView1.Nodes.Add(Root);
```

Now we simply call our subTree function to get the total storage used by the root directory and everything below it. The function returns this as a long integer:

```
long total = subTree(Root);
```

Finally, we set the Root node's text property to indicate this total amount of storage:

```
Root.Text = total.ToString();
}
```

The only problem, of course, is that we don't actually have a function called subTree yet. Creating it is our next task.

THE SUBTREE FUNCTION

As we said above, a recursive program invokes itself in order to perform nested calculations within calculations. That is exactly what we need to calculate the total size of a tree of nested directories. To write a recursive function, you must first imagine that the function is already finished and then consider how the finished function would invoke itself to achieve its aim. You probably need to see this in practice for it to make sense.

The subTree function must start by calculating how much space the files stored directly in the current directory use. To do this, it must first retrieve the DirectoryInfo object stored in the node it receives.

```
private long subTree(
    TreeNode currentNode)
{
    DirectoryInfo currentDir =
        (DirectoryInfo)currentNode.Tag;
    currentNode.Text = currentDir.
        Name + " ";
```

The last line sets the current node's Text property to display the directory name. We can now use the DirectoryInfo object to generate an array of FileInfo objects representing all of the files stored in the directory:

```
FileInfo[] Files =
    currentDir.GetFiles();
```

Once we have this array, a simple foreach loop sums the file sizes in bytes:

```
long total = 0;
foreach (FileInfo f in Files)
{ total += f.Length; }
```

The += operator adds numbers to a running total and is shorthand for total = total + f.Length.

SPACE SAVING

Now we know how much space the files take up, we need to add the space each subdirectory uses. This is similar to the way we added the files. First, we generate an array of DirectoryInfo objects that represents the directories stored in the current directory:

```
DirectoryInfo[] Dirs =
    currentDir.GetDirectories();
```

Each of these subdirectories must be added to the currentNode's Nodes collection. To do this, we use a temporary variable and a foreach loop:

```
TreeNode node;
foreach (DirectoryInfo Dir in Dirs)
{
```

Each TreeNode is created in the same way as the original root directory's node – using the Tag property to store the DirectoryInfo object – and is then added to the current node:

```
node = new TreeNode();
node.Tag = Dir;
currentNode.Nodes.Add(node);
```

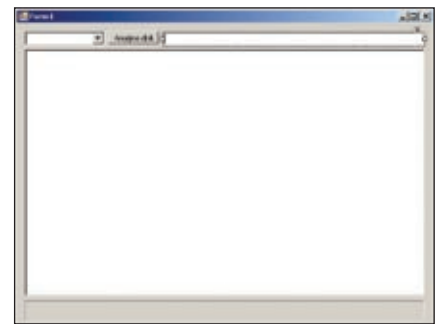
To show the user that the node has been added, so they can track the progress of the disk analysis, we also expand the node:

```
currentNode.ExpandAll();
```

Meanwhile, to keep the PC responsive, we let the system quickly process any pending events:

```
Application.DoEvents();
```

Now we come to the part that many find mysterious. We need to work out the total amount of space the current subdirectory uses.



▲ Add a progress bar below the TreeView and a text box above

This is a simple matter of using the subTree function again:

```
total += subTree(node);
}
```

This is the line that makes most people a little suspicious and bewildered. The reason it works is that every time you call a function, it creates a new copy with its own set of variables and so on. So when subTree calls itself, it isn't really calling itself: it's creating a fresh copy of itself.

The beauty is that this nested function can then call a further nested function, and so on. In this way, subTree creates a chain of copies of itself until eventually it creates a copy that doesn't have any sub-directories to process. This final copy simply counts the space the files use and returns this as its result. This, added to the results of the other finished subTree functions, slowly propagates back up the chain until control returns to the first copy, which returns the final answer. It may be mind-boggling, but it's logical and a very powerful technique. Recursion breaks one very complicated calculation down into simple sums nested inside one another.

Once the foreach loop is complete, the running total gives us the total storage used by the current directory and all the directories below it. This is our final result for this directory and we can add it to its TreeNode's text property:

```
currentNode.Text = currentNode.Text +
    total.ToString();
```

Finally, we return the result as the value of the function:

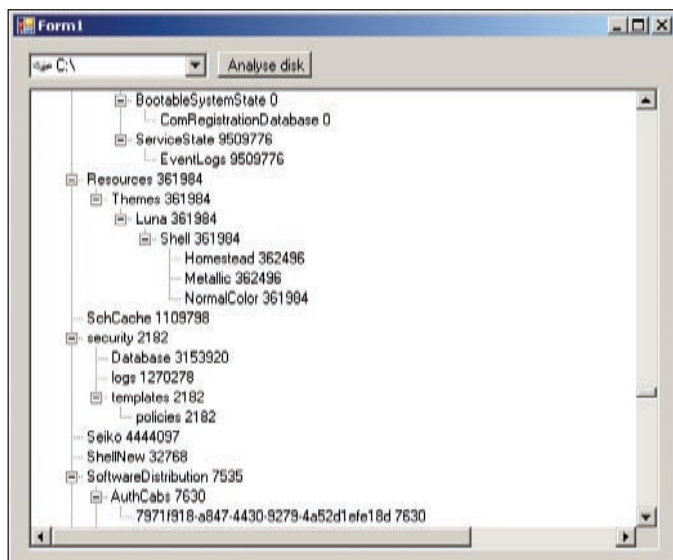
```
return total;
}
```

That's all there is to it. If you run the program, you will see a directory hierarchy build up one directory at a time, as shown on the left.

REFINING THE ANALYSER

We now have the working heart of our disk space analyser. However, we need to make it more user-friendly. One problem is that building up the tree can take time. At the moment, we keep the user informed of progress by redrawing the TreeView control as the program explores each directory. Unfortunately, this redrawing takes more time than it takes to analyse the disk. You can see this is the case by changing the first call to subTree to read:

```
treeView1.BeginUpdate();
long total = subTree(Root);
```



◀ The TreeView shows all the directories on the current disk with their sizes


```
Root.Text = total.ToString();
treeView1.EndUpdate();
```

The BeginUpdate method tells the TreeView control not to bother redrawing itself because we are changing it. To allow the user to see (and manipulate) the finished TreeView, we simply have to call EndUpdate at the right time.

Having made this change, you should find that scanning the disk takes a tenth of the time. This is much more acceptable, but now the user has no idea of what is going on. The solution is to add a progress bar and a text box, as shown in the screen opposite, and update both as our program explores the disk.

The text box shows the name of the directory we're currently analysing. To do this, we simply change the start of subTree to:

```
private long subTree(
    TreeNode currentNode)
{
    DirectoryInfo currentDir =
        (DirectoryInfo)currentNode.Tag;
    textBox1.Text = currentDir.Name;
```

The progress bar is a bit trickier. The length of the bar displayed is controlled by what you store in its Value property. Specify the Maximum and Minimum values, which correspond to a full bar and no bar. The form's constructor is the best place to set up the progressBar:

```
public Form1()
{
    InitializeComponent();
    progressBar1.Minimum = 0;
    progressBar1.Maximum = 100;
}
```

To set the bar length, we need to calculate what percentage of the drive we have analysed. We'll base this on a running grand total of storage space compared with the total amount of space used on the drive. To make this calculation, we first need two global variables:

```
private long grandTotal = 0;
private long driveUse;
```

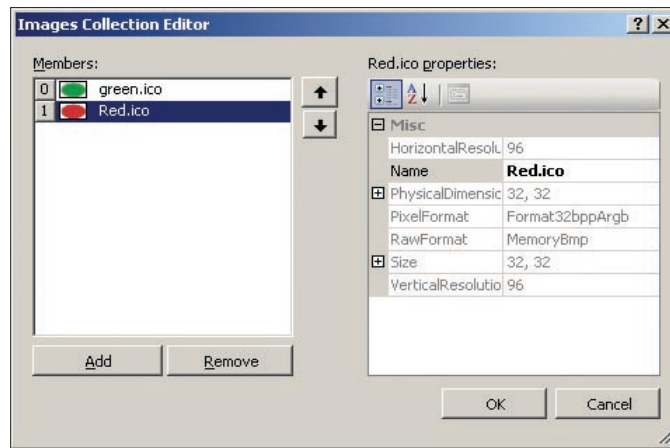
We can find out how much drive space is in use by using a DriveInfo object at the start of the button click event handler:

```
private void button1_Click(
    object sender, EventArgs e)
{
    button1.Enabled = false;
    treeView1.Nodes.Clear();
    string RootDir = dirCombo1.Text;
    DriveInfo Drive =
        new DriveInfo(RootDir);
    driveUse = Drive.TotalSize -
        Drive.AvailableFreeSpace;
    grandTotal = 0;
```

We have included extra lines above to disable the Analyse button temporarily, so the user cannot click it while analysis is in progress, and to clear the TreeView and other variables. We also need to re-enable the button by adding:

```
button1.Enabled = true;
```

We add this at the end of the click event handler.



◀ The blobs ready for use

Modifying subTree to keep track of the grand total of storage analysed is simple enough. After we've summed the file sizes for a given subdirectory, we'll add this to the grandTotal variable:

```
foreach (FileInfo f in Files)
{
    total += f.Length;
}
grandTotal += total;
```

Now that we have the updated grandTotal, it's easy enough to compute the percentage completed and update the ProgressBar:

```
double progress =
    (double) grandTotal /
    (double) driveUse;
if (progress > 1) progress = 1;
progressBar1.Value =
    (int)(progress * 100.0);
```

You may be wondering why we need the If statement. The reason is that the figures for the total space used and the grand total calculated by adding all the file lengths together are not exactly the same. To avoid the problem of doing more than 100 per cent of the task and generating an error, we have to make sure the progress isn't bigger than 1.

ADDING COLOURS

There are plenty of additions you can make to the analysis. You might want to check the file types stored in a directory, say, and characterise the data as programs, documents, music or

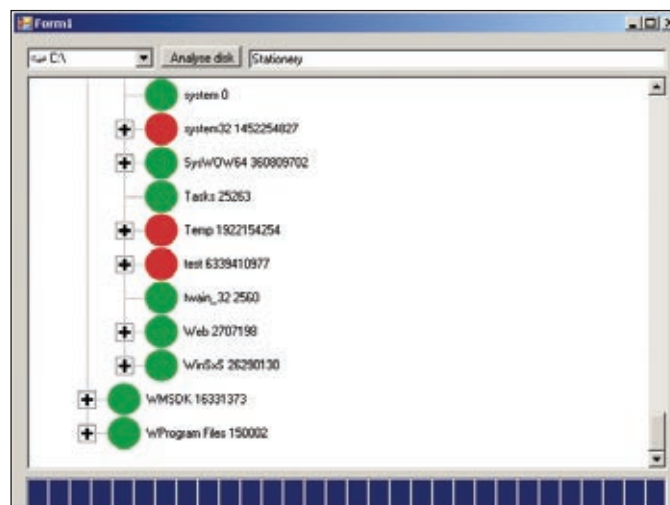
whatever. To show how easy it is to use images with a TreeView control, let's place a red blob next to any directory that has 1GB or more stored in it and a green blob next to smaller directories.

First, we need to store two graphics, red.ico and green.ico, in the program directory. Place an ImageList control on the form and use its custom properties to load both icons; set the green one as index 0 and the red as index 1 (see above).

Next, examine the properties of the TreeView control and set its ImageList property to the name of the ImageList control we have just added, ImageList1, by default. Now all the nodes will show the image each TreeNode's ImageIndex property gives on the right. As this is zero by default, the green blob will show unless we change it. Now we simply add a test to the end of the subTree function that sets the ImageIndex of each node as it is processed:

```
currentNode.Text = currentNode.
    Text +
    total.ToString();
if (total > 1000000000)
{
    currentNode.ImageIndex = 1;
}
return total;
```

When the run is complete, all the big directories are marked with red blobs, as below. You can use this to discover which parts of the directory hierarchy deserve more attention.



◀ Big directories are now clearly identified with red blobs